ИНФОРМАТИКА

УДК 519.68

Д. В. Калинин[1], *М. Ю. Орехов*[1,2]

СПЕЦИАЛИЗИРОВАННАЯ КОНТЕЙНЕРНАЯ БИБЛИОТЕКА ДЛЯ ЗАДАЧ ДИНАМИЧЕСКОГО ОТОБРАЖЕНИЯ ВЕКТОРНОЙ ГРАФИКИ

[1] ФГУП «Научно-исследовательский технологический институт имени А. П. Александрова», Российская Федерация, 188540, Ленинградская обл., г. Сосновый Бор, Копорское шоссе, 72
[2] Санкт-Петербургский государственный университет, Российская Федерация, 199034, Санкт-Петербург, Университетская наб., 7–9

Рассматриваются вопросы инструментальной поддержки разработки системы динамической визуализации векторных объектно-ориентированных 2D схем открытого текстового формата в составе комплекса моделирования сложных технических объектов на уровне проектирования надежной и быстродействующей контейнерной библиотеки. Обеспечение высокой скорости выполнения словарной операции поиска позволяет представлять графические объекты в виде гибких структур — ассоциативных массивов атрибутов переменного типа со строковым индексом, упрощая создание и сопровождение системы отображения в условиях неопределенности перечня типов моделируемых объектов и правил их динамического поведения. Быстрый поиск атрибута по имени приобретает особую важность для функционирования системы в реальном времени при визуализации схем сложных объектов с численностью графических объектов в несколько десятков тысяч. Помимо этого ключевого свойства контейнерная библиотека должна удовлетворять ряду перечисленных в статье специфических требований. Предложен вариант реализации, учитывающий названные ограничения. Продемонстрированы оценки быстродействия операций вставки и поиска для разработанного контейнерного класса и его аналогов из популярных библиотек STL и Qt. Библиогр. 12 назв. Ил. 4.

Ключевые слова: контейнерный класс, гибкая структура, поиск по строковому ключу, ассоциативный массив, векторная графика.

D. V. Kalinin[1], *M. Yu. Orekhov*[1,2]

SPECIALIZED CONTAINER LIBRARY FOR PURPOSES OF VECTOR GRAPHICS DYNAMIC DISPLAYING

[1] Alexandrov Research Institute of Technology, 72, Koporskoe highway, Sosnovy Bor, Leningrad region, 188540, Russian Federation
[2] St. Petersburg State University, 7–9, Universitetskaya nab., St. Petersburg, 199034, Russian Federation

Калинин Дмитрий Владимирович — начальник группы лаборатории систем автоматизации разработки моделирующих комплексов и тренажеров; kdv2112@mail.ru

Орехов Михаил Юрьевич — инженер лаборатории систем автоматизации разработки моделирующих комплексов и тренажеров, аспирант; genazvale2005@yandex.ru

Kalinin Dmitriy Vladimirovich — training and engineering simulators computer-aided design laboratory head of group; kdv2112@mail.ru

Orekhov Mikhail Yurievich — training and engineering simulators computer-aided design laboratory programmer, post-graduate student; genazvale2005@yandex.ru

DOI: 10.21638/11701/spbu10.2016.205

This paper considers instrumental support issues of complex technological objects modelling on the level of container library design for the purposes of vector object-oriented 2D open-text formatted schemes real-time dynamic displaying. Guaranteed rapid key-string search allows presenting the graphical objects as variable structures — string-indexed associative containers of mutable-typed attributes, simplifying visualization system development and maintenance under simulated objects types and dynamic behavior patterns indeterminacy condition. Displaying system efficiency becomes critically dependent from attribute key-string look-up duration as modelled object complexity increases and reveals in great amount of scheme graphical objects, reaching several thousands. Besides this feature, container library must answer some other specific requirements. This article enumerates them and suggests applicable implementation. The present paper also adduces the results of test, estimating insertion and look-up speed, performed for designed container class and its analogues provided by STL and Qt. Refs 12. Figs 4.

Keywords: container class, variable structure, key-string search, associative array, vector graphics.

Введение. Статья посвящена вопросам проектирования надежной и быстродействующей специализированной контейнерной библиотеки. Устройство контейнерного класса представлено как сочетание двусвязного списка, непосредственно *содержащего* размещенные объекты, и набора индексных таблиц поиска объекта по заданному ключу с поддержкой сортировки и фильтрации.

Из накопленного запаса идей, терминов и подходов в области программной реализации словарей — конечных динамических множеств с определенными операциями вставки, поиска и удаления элементов — для индексной таблицы выделены альтернативные структуры данных: хеш-таблица [1, с. 285; 2, с. 323], вариант сбалансированного дерева поиска [1, с. 341], список с пропусками [3] и сплошной массив упорядоченных идентификаторов с функцией бинарного поиска [2, с. 180].

В настоящей работе обоснован выбор массива ключей в качестве индексной таблицы: перечислены решения, позволившие уменьшить время выполнения словарной операции поиска в сравнении с древовидными структурами и обеспечить функциональность сортировки и фильтрации ключей, недоступную для хеш-таблиц.

Примером области приложения контейнерной библиотеки как инструмента разработки является среда динамической визуализации векторных объектно-ориентированных 2D схем открытого текстового формата — компоненты программно-вычислительного комплекса. Одним из условий проектирования среды был учет *неопределенности* палитры типов графических примитивов, объектов и перечня их атрибутов, обусловленной тем, что состав моделируемого оборудования, а также набор специфических свойств каждой единицы могут быть расширены на дальнейших этапах разработки и эксплуатации комплекса. Число размещенных на схеме графических объектов может достигать нескольких тысяч, суммарное количество их графических атрибутов — десятков тысяч. Среда визуализации должна обеспечивать редактирование и динамическое отображение объектов схемы в реальном времени (с частотой обновления не менее 5 FPS). Описание контекста применения среды визуализации приводится в работе [4] и видеоролике [5]. Конкретная задача, обусловившая создание среды в этом контексте, сформулирована в статьях [6, 7].

В п. 1 данной статьи предложен способ организации *разнородных* атрибутов графического объекта в виде контейнерного класса. Характерные для области приложения требования к выбору структур данных его реализации перечислены в п. 2. Модель контейнера, удовлетворяющая названным требованиям с гарантией высокого быстродействия, описана в п. 3. В п. 4 представлены сравнительные результаты

тестов оценки времени выполнения операций вставки и поиска для разработанного контейнерного класса и его аналогов из библиотек STL и Qt.

1. Графический объект — контейнер атрибутов. В рамках использования открытого текстового формата схемы графические примитивы и объекты задаются SVG-подобными определениями собственных атрибутов:

```
CLASS=obj  TYPE=Std_ana_veu NAME= DESCR=МощнПотребл X=2102 Y=209 Z=822 ...
CLASS=prim TYPE=ellipse NAME= DESCR=Обмотка!32трансформатора X=2610.42 ...
```

Рассмотрим подход к построению иерархии графических сущностей с неопределенным набором свойств (примитив, составной объект, схема) в виде объектов *гибкой* структуры — ассоциативных контейнеров: таблиц идентификаторов атрибутов произвольного типа со строковым индексом-ключом поиска, формируемых в процессе разбора текстового определения. Под гибкостью структуры здесь подразумевается возможность изменения как количества элементов, так и их качества — тип хранимого значения атрибута (строковый, целочисленный или вещественный) определяется последней операцией присваивания. Класс «таблица идентификаторов» имеет вид

```
template<class Attr> class IdentifierTable {
public:   //Подстрока SubString - универсальный аргумент функций строковой
          Attr &operator()(const SubSting &AttrName);  // библиотеки [8, 9].
    const Attr &operator()(const SubSting &AttrName) const;
    ...
    virtual SString generateDefinition() const;   // Сохранение определения.
    virtual void    loadDefinition(const SubString &textFlow);  // Загрузка.
};
```

Оператор «()» гибкой структуры, соответствующий оператору «.» вычисления смещения поля в жесткой, реализует доступ к полю по строковому индексу. Он вызывает методы таблицы идентификаторов для поиска/создания атрибута по его имени.

Класс «примитив» используется для представления всей палитры примитивов (линия, прямоугольник, эллипс и др.) базовой графической библиотеки:

```
class Primitive : public IdentifierTable<Attribute> {
    PlatformDependentPrim *graphicsItem; // Платформенно-зависимое
    ...                                  // представление примитива.
};
```

Составной графический объект (через посредство «примитива») и схема наследуют функциональность `IdentifierTable` для перечисления собственных атрибутов и генерации/загрузки текстовых определений, а также содержат списки образующих их объектов и примитивов.

Предложенный подход позволяет обеспечить:

1) поддержку модификаций набора свойств моделируемого объекта, а также динамическое создание и удаление «пользовательских» полей и смену их типа;

2) независимость от реализации и текущей версии графической платформы. Переход к иной платформе разработки среды визуализации может повлечь изменение номенклатуры наличных графических примитивов и их свойств. Взаимодействие *абстрактного* контейнера со своим платформенно-зависимым представлением *жесткой* структуры ограничивается последовательным присваиванием значений атрибутов, инкапсулированным в процедурах *драйвера* объекта. Таким образом, замена

платформы отразится лишь в необходимой правке имеющихся и вероятном введении новых драйверов согласно расширению набора примитивов;

3) возможность создания окон графического интерфейса без учета специфики объектов воспроизведения. Рассмотрим пример окна свойств графического объекта: при применении контейнерной модели достаточно реализовать одну функцию обновления окна, считывающую имена и значения атрибутов в цикле, вместо нескольких функций, обращающихся к полям каждой из сущностей жесткой конфигурации;

4) отвлечение механизма динамического обмена от особенностей графических объектов. Этот механизм проецирует величину моделируемого параметра на значение графического свойства для управляющих параметров и осуществляет обратное преобразование для управляемых. Использование абстрактного интерфейса именованного доступа к атрибутам позволяет обобщить процедуру преобразования.

2. Требования к проектированию. Основным и критическим требованием к предлагаемой реализации, гарантирующим высокое быстродействие, являются малые временные издержки *поиска* по строковому ключу. Это условие становится очевидным на примере динамического отображения схемы с большим числом объектов. Пусть для визуализации расчета некоторой моделирующей задачи из общего числа графических атрибутов объектов необходимо модифицировать 10 000. Тогда для воспроизведения схемы с частотой 5 FPS следует 5 раз в секунду выполнить составную операцию обновления, несколько раз включающую поиск 10 000 имен в различных контейнерах.

На скорость поиска по строковому индексу помимо особенностей самого контейнера оказывает влияние время работы оператора *сравнения* ключей. Предполагается использование контейнерной библиотеки совместно со специализированной строковой библиотекой, где операции сравнения и копирования строк близки по быстродействию к соответствующим низкоуровневым функциям стандартной библиотеки [8, 9].

В силу характера используемых данных контейнер должен поддерживать *дублирование* ключей, а также группирование одноименных элементов с доступом по ключу и целочисленному индексу. К нескалярным графическим свойствам относятся координаты вершин многоугольника, компоненты цвета, точки градиента и т. д.

Для обеспечения разнообразия форм представления графической информации обязательно оснащение контейнерной библиотеки средствами *сортировки* и *фильтрации* данных по заданному, вероятно, составному критерию. С наличием таких инструментов пользователь среды отображения приобретает возможность выбрать атрибуты имеющихся объектов схемы, наложить фильтры и установить направление сортировки по *каждому* из них, чтобы в полученном перечне внести контекстные изменения, скажем, в маркировках моделируемого оборудования.

Учитывая предполагаемые масштабы задачи визуализации и большое число итераций циклов вставки и удаления при формировании контейнеров графических объектов, необходимо требовать *минимизации* создания временных и служебных объектов, заключающих хранимые данные, поскольку обилие запросов выделения блоков памяти разных размеров способствует фрагментации кучи.

3. Описание реализации. В качестве схематической модели самого общего устройства контейнерного класса рассматривается двусвязный список, дополненный набором индексных таблиц. Список-владелец содержит данные *непосредственно*, имеет постоянное время вставки в начало и в конец. Каждая из таблиц обеспечивает минимальное время поиска элемента по заданному ключу, а также упорядочение и вывод данных по некоторому правилу. В п. 3.1–3.7 представлен вариант реализации

содержания предложенной формы, передающего ей необходимые свойства для соответствия перечисленным ранее условиям. На рис. 1 приведена иерархия описываемых программных классов в графической нотации UML со спецификацией внешних и внутренних интерфейсов.

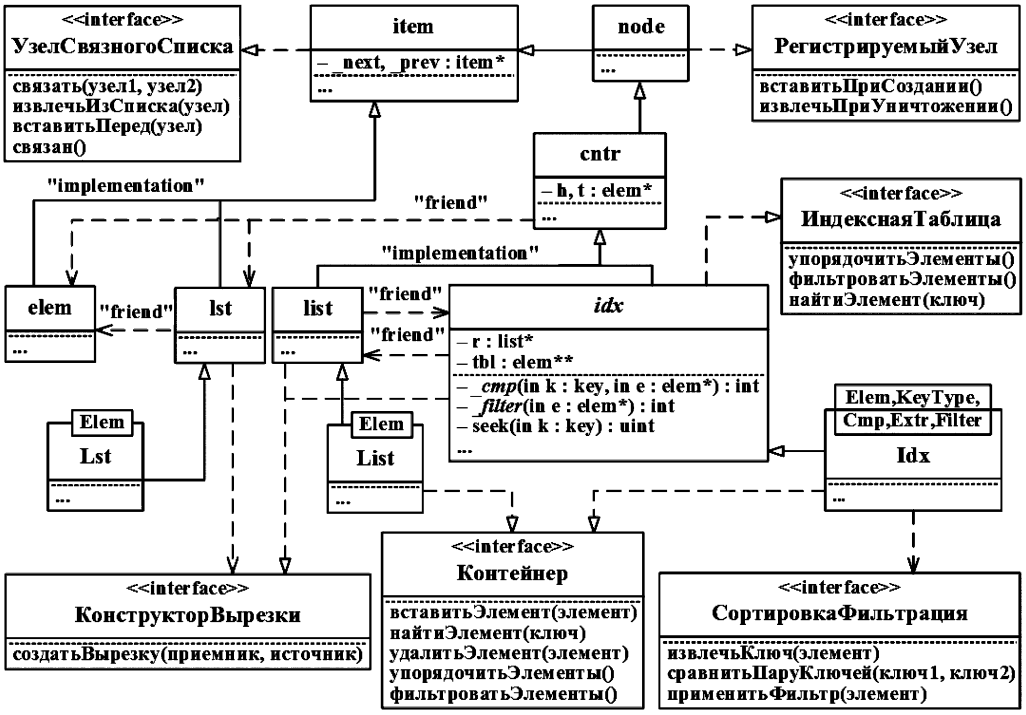

Рис. 1. Иерархия программных классов

3.1. Абстрактный узел списка item. Требование минимального употребления вспомогательных объектов при размещении данных выполняется, если все потенциально вносимые в контейнер сущности являются потомками абстрактного «узла» двусвязного списка. Этим помимо унификации размеров выделяемых блоков уменьшается число обращений к динамической памяти ввиду отсутствия потребности в служебных объектах. Принципиальная функциональность узла инкапсулируется в классе `item`:

```
class item {
    mutable item *_next,*_prev;
protected:
    static void link(const item *i)               // Замкнуть на себя.
        { i->_prev=i->_next=(item*)i; }
    static void link(const item *p,const item *n) // Связать аргументы.
        { (p->_next=(item*)n)->_prev=(item*)p; }
    static void unlink(const item *i)             // Извлечь из списка.
        { (i->_prev->_next=i->_next)->_prev=i->_prev; }
    static void ins(const item *m,const item *i)  // Вставить i перед m.
        { link(m->_prev,i);link(i,m); }
```

```
    static int nol(const item *i) { ... } // Элемент не связан (замкнут).
public:
    item() { }
    item(const item &) { link(this); }    // Замкнуть созданную копию.
    void operator=(const item &) { }
    virtual ~item() { }
    ...
};
```

3.2. База наследования элементов контейнера elem. Производство элементов от общего предка допускает совместное хранение объектов разных уровней иерархии. Наследование с ключом `private` делает «происхождение» от `item` скрытым как для потомков `elem`, так и для внешних классов за исключением дружественных:

```
class elem : private item {
    friend class lst; friend class cntr;
public:
    elem()            : item(*this) { } // Новый элемент замкнут.
    elem(const elem&) : item(*this) { } //
    ~elem() { if(!item::nol()) err("linked somewhere"); } // Аварийный выход
    ...                         // при попытке деструкции связанного элемента.
};
```

3.3. Список временного хранения lst. Описываемая реализация линейного двусвязного списка предполагает *владение* элементами для предотвращения их многократного уничтожения: два контейнера не могут иметь общих элементов, деструкция контейнера предусматривает деструкцию данных. Поэтому передача части элементов между списками-владельцами возможна либо созданием «глубокой» копии, либо путем вырезки. Вырезки *перемещаемых* (move) элементов организуются в виде кольцевых списков промежуточного хранения — объектов класса `lst`:

```
class lst : private item { // Головной элемент списка.
    friend class cntr;
    mutable uint len;      // Число элементов.
    ...
    static void ins(const item *m,const elem *e); // Вставить e перед m.
    static uint ins(const item *m,const lst &l);  // Вставить содержимое l
protected:                                        // перед m. Замкнуть l.
    struct _cut_p { };                       // Абстрактная граница вырезки.
    typedef void _cut_f   (lst *out,_cut_p *from); // Методы вырезки в out.
    typedef void _cut_to_f(lst *out,_cut_p *from,const _cut_p *to); //
    ...
    lst &operator=(const lst &l);        // Переместить (move) содержимое l.
    lst &operator<<(elem *e) { ins(this,e);++len;return *this; } // Вставка
    lst &operator<<(const lst &l);                          // в конец списка.
public:
    lst() : item(*this) { len=0; }
    lst(const lst &l); // Переместить содержимое l.
    // «Виртуальные» конструкторы lst.
    lst(_cut_f *fn,_cut_p *from)                    { fn(this,from); }
```

```
    lst(_cut_to_f *fn,_cut_p *from,const _cut_p *to) { fn(this,from,to); }
    ~lst(); // Деструкция содержимого.
    ...
};
```

Возможность вырезки элементов из заданного источника обеспечивается применением механизма «виртуального» (фактического) конструирования, избавляющего от необходимости включения в состав `lst` специфических конструкторов для каждого вероятного потребителя: конкретный способ вырезки сообщается экземпляру `lst` в момент создания параметрически. Механизм использует *стандартное* свойство языка [10] опускать промежуточные конструкторы, поддерживаемое оптимизирующим компилятором. Вызов «виртуальных» конструкторов разрешен объектам классов, имеющих доступ к определению абстрактных границы и методик вырезки.

Контроль типов хранимых элементов осуществляется введением шаблона `Lst` с опубликованными операторами присваивания и вставки. Так, конкретная инстанция `Lst` может содержать потомков `elem` одного уровня и ниже, попытка же размещения в `Lst` более абстрактного представителя иерархии будет блокирована компилятором:

```
template <class Elem> class Lst : public lst {
public:
    Lst &operator=(const Lst &l) { return (Lst&)lst::operator=(l); }
    Lst &operator<<(Elem *e)     { return (Lst&)lst::operator<<(e); }
    ...
};
```

3.4. База наследования структур данных контейнера cntr. Каждый список-владелец элементов контейнера ассоциируется с одной или несколькими индексными таблицами. Для упрощения взаимодействия этих сущностей и управления ими предлагается использовать единую базу наследования — `node`, что позволяет организовывать индексные таблицы в кольцевой список вокруг головного элемента — списка контейнера — с автоматической регистрацией при создании и извлечением при деструкции:

```
class node : public item { // Потомкам «известно» о происхождении от item.
public:
    node()              : item(*this) { } // Головной элемент списка.
    node(const node &n) : item() { ins(&n,this); } // Вставка перед n.
    ~node( ) { unlink(this); }            // Деструкция с извлечением.
    node &operator=(const node &n)        // Смена списка.
        { unlink(this);ins(&n,this);return *this; }
    ...
};
```

Класс `cntr`, как *общий* предок линейного двусвязного списка-владельца элементов контейнера и индексной таблицы, поддерживает константные (RO — read-only) и неконстантные (RW — read-write) итераторы. RO-итератор на время своего существования блокирует модификации объекта `cntr`, увеличивая счетчик `cntr::lock`. RW-итераторы, порожденные от `node`, формируют кольцевой список для автоматической проверки и коррекции их позиций при вставке и вырезке элементов экземпляра `cntr`. Попытка вырезки итерируемого фрагмента списка приводит к *явному* аварийному завершению работы, чем обеспечивается *надежность* итерирования:

```
class cntr : protected node { // Элемент кольцевого списка.
    void init() { lock=0; len=0; h=t=NULL; }
protected:
    node itrs;         // Головной элемент кольцевого списка RW-итераторов.
    mutable uint lock; // Счетчик блокировок модификаций.
    uint len;          // Число хранимых элементов.
    elem *h,*t;        // Граничные элементы («до первого» и «за последним»).
    typedef lst::_cut_p _cut_p; // Поддержка «виртуального» конструирования
    typedef lst::_cut_f _cut_f; // lst для потомков и RW-итераторов.
    typedef lst::_cut_to_f _cut_to_f; //
    ...
    cntr() : node() { init(); }   // Головной элемент кольцевого списка.
    cntr(const cntr &n) : node(n) { init(); } // Элемент списка перед n.
public:
    class __itr; friend class __itr; // База наследования итераторов.
    ...
};
```

Дополнительным преимуществом проектирования структур данных контейнера на едином основании является возможность выделения общих свойств и методов константных и неконстантных итераторов списка и индексной таблицы в корне иерархии наследования — классе `cntr::__itr`:

```
class cntr::__itr {
    friend class cntr;
protected:
    elem *c;         // Текущий элемент.
    cntr *r;         // Итерируемая структура данных.
    mutable uint cp; // Текущая позиция.
    ...
    __itr(const cntr &l)  { *this=l; } // К элементу h.
    __itr(const __itr &m) { *this=m; } // К элементу m.c.
    void operator=(const cntr &l)  { c=l.h;r=(cntr*)&l;cp=0; }
    void operator=(const __itr &m) { c=m.c;r=m.r;cp=m.cp; }
};
```

3.5. Список элементов контейнера list. Функциональное содержание индексируемого итерируемого двусвязного линейного списка, реализованное в классе `list`, гарантирует автоматическую реиндексацию таблиц своих элементов и правку позиций RW-итераторов в методах вставки и вырезки. Контейнер `list` определяет собственные итераторы через общую базу `list::_itr`, дополняющую возможности `cntr::__itr` методами инкремента, смещения и т. п. Контроль типов хранимых элементов обеспечивается на уровне шаблона `List` с открытыми операторами присваивания и вставки:

```
class list : private cntr { // Головной элемент списка индексных таблиц.
    friend class idx;
    class _itr; friend class _itr; // База наследования итераторов.
                                   // : protected cntr::__itr.
    void init(uint sz); // Разместить и связать элементы h и t размером sz.
```

```
protected:
    static void _cut(lst *out,list *l); // Вырезать все элементы.
    list &operator=(const lst &l) { if(len)del();return *this<<l; }
    list &operator<<(elem *e);          // Вставка в конец списка.
    list &operator<<(const lst &l);     // Присвоить содержимое l.
public:
    class citr; friend class citr;      // Декларация RO-итератора.
    class itr;  friend class itr;       // Декларация RW-итератора.
    ...
    list(uint sz=sizeof(elem)) : cntr() { init(sz); }
    list(const lst &l,uint sz=sizeof(elem)); // Заимствовать элементы l.
    ~list();                            // Деструкция элементов.
    ...
    lst cut() { return lst((_cut_f*)_cut, (_cut_p*)this); } // Вырезать.
    list &del();                       // Вырезав, уничтожить содержимое.
};
```

Неконстантный итератор `list::itr` определяет операторы вставки за текущий элемент и ряд методов «виртуального» конструирования хранилищ вырезок `lst`:

```
class list::itr : private node, private _itr {
    friend class citr; // citr «известно» о происхождении itr от _itr.
protected:
    // Методы «виртуального» конструирования lst. Вырезать
    static void _cut (lst *out,itr *m);      // элемент, сместиться вправо.
    static void _cut_rest(lst *out,itr *m);  // элементы правее включительно.
    static void _cut_to(lst *o,itr *m,const itr *n); // элементы правее до n,
    ...                                              // включая текущий.
    itr &operator<<(elem *e);                // Вставить правее, сместиться
    itr &operator<<(const lst &l);           // к последнему вставленному.
public:
    itr(list &l)      : node(l.itrs),_itr(l) { } // Регистрация в списке
    itr(const itr &m) : node(m),     _itr(m) { } // при создании
    itr &operator=(list &l);                     // и смене контейнера.
    ~itr() { }                          // Извлечение из списка итераторов.
    lst cut() { return lst((_cut_f*)_cut,(_cut_p*)this); } // Методы вырезки.
    ...                                                    //
    lst cut_to(const itr &n)                               //
       { return lst((_cut_to_f*)_cutr_to,(_cut_p*)this,(const _cut_p*)&n); }
};
```

3.6. Индексная таблица списка контейнера idx. Перечислим возможные варианты реализации индексной таблицы, поддерживающие дублирование ключей:

• хеш-таблица;

• популярные при проектировании ассоциативных контейнеров структуры данных: красно-черное дерево [1] и список с пропусками [3], используемые разработчиками библиотек STL (`std::multimap`) и Qt (`QMultiMap`) соответственно [11, 12], гарантирующие вставку и поиск с логарифмическими затратами времени в худшем случае;

• *сплошной* упорядоченный массив с бинарным поиском объектов: при логарифмическом порядке роста временных издержек поиска в худшем случае время вставки будет меняться линейно: сортировка потребует перемещения элементов вправо.

Отказ от применения хеш-таблиц мотивируется, во-первых, принципиальной невозможностью представления ключей в отсортированном виде с сохранением эффективности выполнения словарных операций и, во-вторых, высокими издержками обращения с составным ключом. Рассмотрим пример поиска записи в таблице по ключу, образованному строковыми значениями нескольких полей. В этом случае для использования хеш-функции от строкового ключа, поставляемой с реализацией хеш-таблицы, необходима конкатенация значений данных полей с неизбежным обращением к динамической памяти для создания и уничтожения буфера результирующей строки. Альтернативой объединению строк является дооснащение реализации собственными *качественными* хеш-функциями генерации индекса для *каждой* комбинации полей ключа поиска.

Основанием выбора сплошного массива служит его превосходство по результатам теста быстродействия операции поиска, приведенным в п. 4. Здесь же назовем ряд факторов, обеспечивающих этот результат:

• возможность хранения *индекса* последнего найденного элемента в качестве начальной позиции повторного поиска;

• *меньшее* по отношению к древовидным структурам число операций сравнения в худшем случае, равное $\lfloor \log_2(n) \rfloor + 1$ [2];

• *локализованное* размещение как самого массива, так и объектов списка, находящихся близко друг к другу в виду малой фрагментированности кучи, позволяющее более эффективно использовать кэш процессора. В то же время издержки операции разыменования указателей на служебные объекты («узлы») древовидной структуры, расположенные на страницах памяти *разреженно*, при интенсивном поиске оказываются весьма высокими.

Отметим принципиальные особенности разработанной индексной таблицы. Класс `idx` включает массив указателей на элементы контейнера. Крайние элементы массива хранят указатели на граничные элементы списка. Состав и очередность указателей в массиве определяются согласно заданным условиям фильтрации и сортировки. Наборы элементов с равными ключами образуют в массиве группы дубликатов.

Положение элемента контейнера в индексной таблице вычисляет функция бинарного поиска с непосредственной передачей ключа поиска. Функция возвращает позицию первого найденного элемента списка с заданным ключом, кэшируя ее в поле `idx::found`, либо 0, если поиск оказался неуспешным. В случае же неудачного поиска в `found` записывается индекс элемента с ближайшим *снизу* ключом. В дальнейшем значение `found` используется как начальная позиция бинарного поиска.

Таблицы `idx`, имея предком класс `node`, при создании связываются в кольцевую структуру для обновления в обработчиках модификации списка элементов контейнера. Наследование `idx` от `cntr` обеспечивает поддержку константных и неконстантных итераторов произведением их от общей базы `cntr::__itr`, делает возможным применение механизма блокировки на время реиндексации и константного итерирования, предоставляет доступ к «виртуальным» конструкторам промежуточных хранилищ `lst`. Процедура вырезки из таблицы подразумевает формирование связанной последовательности извлекаемых элементов основного списка в заданном таблицей *порядке*:

```
class idx : private cntr {
    friend class list;
protected:
    idx(list &l);
    struct key { }; // Абстрактный ключ поиска.
    ...             // Доступ к «виртуальным» конструкторам lst.
private:
    list *r;    // Основной список контейнера.
    elem **tbl; // Упорядоченный массив указателей на элементы контейнера.
    mutable uint found; // Кэшируемый индекс найденного элемента.
    class _itr; friend class _itr; // База наследования итераторов
                                   // : protected cntr::__itr.
    // Методы сортировки и фильтрации.
    virtual int _cmp(const key &k,const elem *e) const = 0;
    virtual int _filter(const elem *e)           const = 0;
    uint seek(const key& k) const; // Функция бинарного поиска.
    // Обновить таблицу при модификации основного списка:
    void insx(elem *p,elem *n); // вставить
    void cutx(elem *p,elem *n); // удалить последовательность (p,n).
           // Методы вырезки. //
    static void __cut(lst *out,idx *x,uint start,uint end);
    ...
protected: //
    static void _cut (lst *out,idx *x) { __cut(out,x,1,x->len); }
    ...
public:    //
    lst cut() { return lst((_cut_f*)_cut, (_cut_p*)this); }
    ...
    class citr; friend class citr; // RO-итератор : private _itr.
    class itr;  friend class itr;  // RW-итератор : private node,
                                   //               private _itr.
};
idx::idx(list &l) : cntr(l), r(&l), found(0) {
    h=(elem*)l.head(); t=(elem*)l.tail();
    tbl=new elem**[2]; tbl[0]=h; tbl[1]=t;
}
```

3.7. Средства сортировки и фильтрации. В теле функции бинарного поиска вызывается метод сравнения аргумента с содержащимся в элементе контейнера ключом. Поскольку способ извлечения ключа из элемента на этом уровне абстракции не задан, метод сравнения `idx::_cmp` является чисто виртуальным:

```
uint idx::seek(const key &k) const {
    register uint i; register int c;
    if((i=found)!=0u) {     // Начать поиск с кэшируемого значения.
        if((c=_cmp(k,tbl[i]))==0) return i;  // Совпадение найдено.
        if(c<0) ... // Назначить границы и перейти
    } ...           // к циклу бинарного поиска.
}
```

Проверка адекватности условию фильтрации выполняется при обработке добавления элементов в основной список чисто виртуальным методом `idx::_filter`:

```
void idx::insx(elem *p,elem *n) {  // В список вставлена вырезка (p;n).
    elem *i=p;
    for( ; (i=list::next(i))!=n; ) {
        if (!_filter(i)) continue; // Анализ выполнения условия.
        ...                        // Поиск позиции для вставки в таблицу.
    }
}
```

Конкретное содержание абстрактные члены получают в составе шаблона `Idx`, которому при создании кроме типа элемента контейнера сообщаются тип ключа поиска, а также типы функциональных классов, устанавливающих методику сравнения двух ключей, правило извлечения ключа из элемента контейнера и критерий фильтрации:

```
template <class Elem,class Key,class Cmp,class Extr,class Filter>
class Idx : public idx {
    virtual int _cmp(const key &k,const elem *e) const
        { return _cmp_( (const Key&)k,_extr_(*((Elem*)e)) ); }
    virtual int _filter(const elem *e) const
        { return _filter_( *((Elem*)e) );
public:
    Cmp    _cmp_;
    Extr   _extr_;
    Filter _filter_;
    ...
};
```

Для описания функциональных классов можно воспользоваться макросами:

```
#define defComparer(id, expr) struct id { template <class Key1,class Key2> \
    int operator() (const Key1 &k1,const Key2 &k2) const { return expr; } }
#define defExtractor(id, type, expr) struct id { template <class Elem> \
    type operator() (const Elem &e) const { return expr; } }
#define defFilter(id, expr) struct id { template <class Elem> \
    int operator() (const Elem &e) const { return expr; } }
```

Проиллюстрируем работу средств сортировки и фильтрации:

```
struct Attribute : public elem { // Сущность для хранения.
    Attribute(const SString &name,int val): _name(name), _value(val) { }
    SString _name;
    int    _value;           // Значение - целое, методы смены типа опущены.
};
defExtractor(AttrKey,const SubString&,subString(e._name); // Строковый ключ.
defComparer(AttrCmp,compare(k1, k2));   // Сравнить ключи лексикографически.
defFilter(AttrFilter,e._value > 5);     // Значение > 5.
typedef Idx<Attribute, SubString, AttrCmp, AttrKey, AttrFilter> AttrTab;

int main(int argc, char** argv) {
```

```
    List<Attribute> attributes;  // Контейнер атрибутов.
    AttrTab attrTab(attributes); // Индексная таблица.
    for (int i = 10; i > 0; --i) // Сформировать список.
        attributes << new Attribute(‘‘Attr#’’ / i2a(i), i); // Attr#i
    for (AttrTab::citr i(attrTab); !(++i)->eol();) // Содержимое таблицы.
        std::cout << i->_name << ‘‘_’’ << i->_value << ‘‘\t’’;
    // Вывод печати: Attr#10_10 Attr#6_6 Attr#7_7 Attr#8_8 Attr#9_9
    return 0;
}
```

4. Демонстрация быстродействия операций вставки и поиска. Тест определения скорости вставки в контейнерный класс представляет собой измерение времени циклического добавления объекта `Attribute` в связный список и указателя на него в индексную таблицу. При испытании контейнеров библиотек STL и Qt использовались соответствующие сочетания связных списков `std::list<Attribute>` и `QLinkedList<Attribute>` с индексными таблицами `std::multimap<SString,Attribute*>` и `QMultiMap<SString,Attribute*>`. Тест выполнен для компилятора gcc4.4.0. Метки на оси абсцисс обозначают размер массива входных данных, ординаты точек кривых фиксируют результат в секундах. В обоснование числовых параметров эксперимента отметим, что указанные размеры массива данных характерны для прикладной задачи динамической визуализации, а именно для таблицы моделируемых параметров расчетной задачи.

Время вставки 150 000 элементов в список `attributes` и таблицу `attrTab` разработанного контейнерного класса в худшем (ключи поиска расположены по убыванию) и в среднем случае (данные частично упорядочены по возрастанию — первыми следуют элементы с четным индексом в ключе, затем остальные) оказывается относительно *наибольшим* (рис. 2). Если же входной поток информации сортирован по возрастанию, длительность вставки — *наименьшая* (рис. 3). При анализе этих оценок необходимо учитывать специфику вставляемых графических данных, формируемых при разборе строк *частично упорядоченных* атрибутов: текстовые определения графических объектов могут генерироваться с перечислением атрибутов по алфавиту, т. е. по возрастанию.

Комплексный тест быстродействия поиска в каждой из пяти итераций цикла производит пятикратный поиск в контейнере данного размера каждого элемента с извлечением его значения. Время поиска в `attrTab` является относительно *наименьшим* безусловно (рис. 4). В зависимости от страничного размещения элементов, определяемого очередностью создания, и степени упорядоченности последовательности разыскиваемых ключей время работы функции бинарного поиска индексной таблицы описывается одним из графиков, заключенным между кривыми `AttrTabBest` и `AttrTabWorst`. Кривые соответствуют максимальной и минимальной эффективностям программного (на уровне `idx::seek`) и аппаратного (на уровне процессора) кэширования.

Полученные результаты согласуются с теоретическими характеристиками тестируемых структур данных. Ордината точки каждой представленной кривой есть измерение времени работы цикла, т. е. сумма измерений времени выполнения рассматриваемой словарной операции для контейнера, размерностью равной ее абсциссе. Размерность итеративно изменяется во внешнем цикле от 1000 элементов с шагом 1000. Время вставки в упорядоченный массив в худшем случае имеет линейный порядок

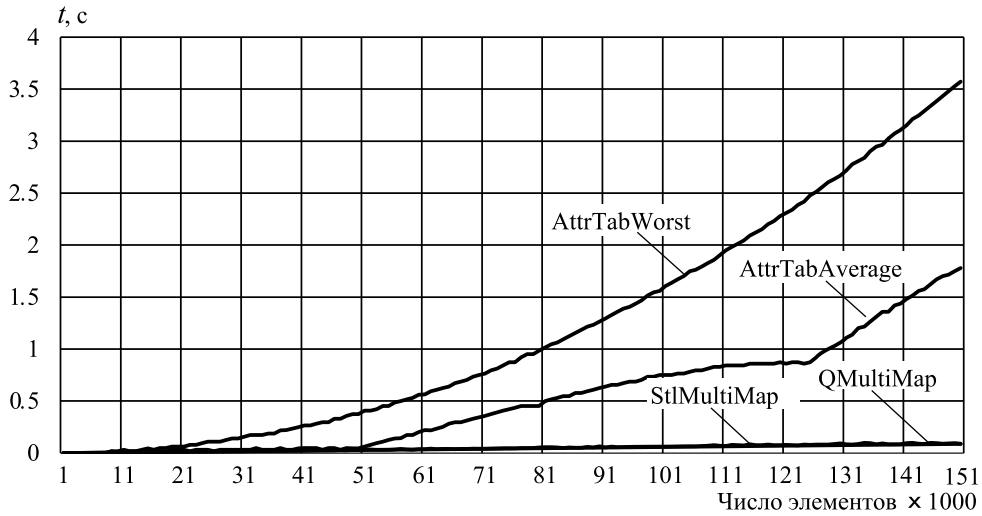

Рис. 2. Оценки быстродействия операции вставки в худшем и среднем случаях

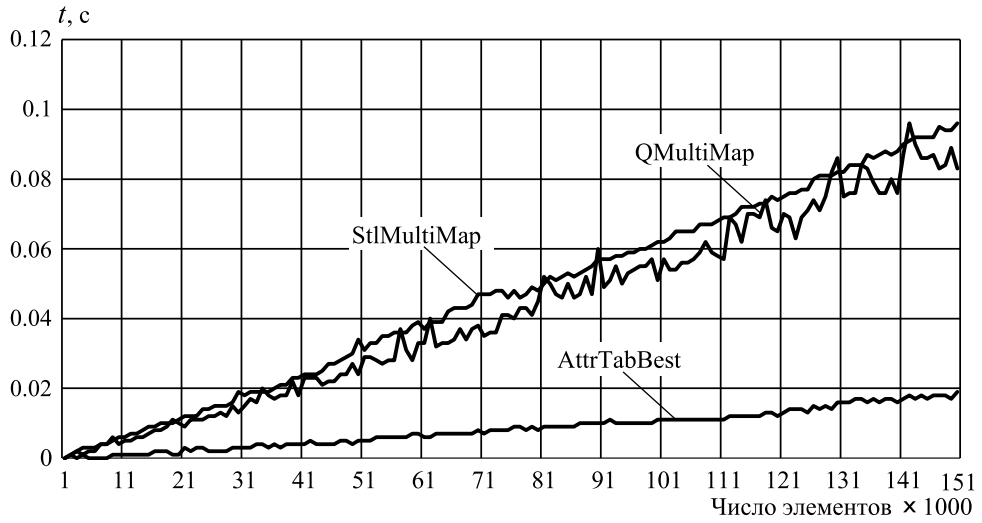

Рис. 3. Оценки быстродействия операции вставки в лучшем случае

роста — $O(n)$, так как требует перемещения всех элементов вправо на каждой итерации. Таким образом, если обозначить абсциссу точки кривой `AttrTabWorst` как n, ее ордината с учетом времени поиска ключа должна иметь зависимость

$$\sum_{i=1}^{n}(i + \lfloor \log_2 i \rfloor + 1) \approx \sum_{i=1}^{n} i = n \cdot (n+1)/2,$$

т. е. изменяться квадратично, что и видно на рис. 2. Словарные операции древовидных структур теоретически относятся к логарифмическому классу эффективности — $O(\log n)$ [1, 2]. Это означает, что ордината точки кривой `StlMultiMap` с абсциссой n

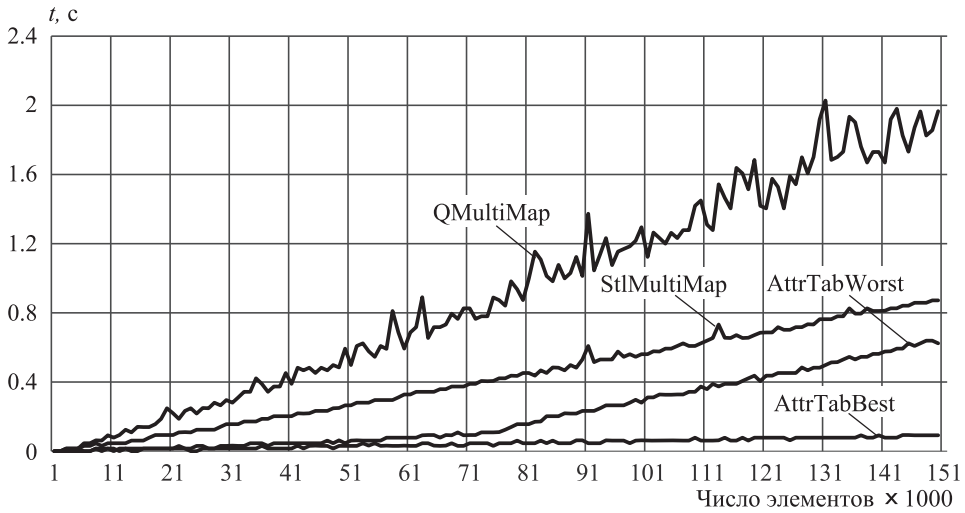

Рис. 4. Оценки быстродействия операции многократного поиска

должна выражаться как

$$\sum_{i=1}^{n} \log_2 i = \log_2 n!,$$

т. е. изменяться логарифмически, что и отражено на рис. 2–4.

Учитывая полученные результаты, простоту организации структур данных и выполнение предъявленных условий, можно сделать вывод о целесообразности выбора упорядоченного массива для реализации индексной таблицы. Применение созданной контейнерной библиотеки в качестве одного из основных инструментов разработки графической среды визуализации предоставляет функциональные возможности:

• правки векторных графических 2D схем визуализации результатов расчета моделирующих задач в редакторах схем, объектов, свойств и групп свойств с поддержкой операций отмены и повтора модификаций (undo/redo);

• хранения разработанных пользователем графических объектов (схемы, графические объекты, связи) и наборов их свойств (цвета, кисти, перья, шрифты) в соответствующих библиотеках с каскадным обновлением схем при изменениях библиотек;

и гарантирует технические характеристики:

• передачи схемам управляющих воздействий — обеспечения их динамического отображения на компьютере «офисной» архитектуры с частотой до 50 FPS;

• времени загрузки насыщенных схем (более 1000 объектов) — 0.3–0.7 с.

Заключение. Сформулированы требования к проектированию надежного и быстродействующего специализированного контейнерного класса, применимого в задачах динамического отображения векторных схем большого числа моделируемых объектов. Обоснован выбор структур данных, отмечены ключевые аспекты варианта реализации. Приведены оценки быстродействия операций вставки и поиска для описанного контейнера в сравнении с его аналогами из библиотек STL и Qt.

Литература

1. *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ. 3-е изд. / пер. с англ. И. В. Красикова. М.: Издат. дом «Вильямс», 2013. 1328 с. (*Cormen T., Leiserson C., Rivest R., Stein C.* Introduction to Algorithms)

2. *Левитин А.* Алгоритмы: введение в разработку и анализ / пер. с англ. С. Г. Тригуб, И. В. Красикова. М.: Издат. дом «Вильямс», 2006. 576 с. (*Levitin A.* Introduction to the Design & Analysis of Algorithms)

3. *Pugh W.* Skip Lists: A Probabilistic Alternative to Balanced Trees // Communications of the ACM. 1990. Vol. 33, N 6. P. 668–676.

4. *Безлепкин В. И., Кухтевич В. О., Образцов Е. П., Мигров Ю. А., Шаленинов А. А., Деулин А. А.* Программно-технический комплекс «Виртуальный энергоблок АЭС с ВВЭР» (ПТК «ВЭБ») для проверки проектных решений АЭС-2006. [Электрон. ресурс] URL: http://www.gidropress.podolsk.ru/fi-les/proceedings/mntk2013/documents/mntk2013-168.pdf (дата обращения: 21.10.2015).

5. Горизонты Атома (16.02.2013): Виртуальная АЭС // Информ. агентство «Российское атомное сообщество». [Электрон. ресурс] URL: http://www.atomic-energy.ru/video/40025 (дата обращения: 15.09.2015).

6. *Калинин Д. В.* Многофункциональный графический редактор видеокадров БПУ в составе моделирующего комплекса «Виртуальный энергоблок». [Электрон. ресурс] URL: http://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxzaGFyaW5nc3BhY2UxNnxneDo0YWI3NTRhM2JlZTMxODE5 (дата обращения: 20.02.2016).

7. *Orekhov M. Yu.* Specialized Container and String Systems Employment in Vector Graphics Environment Development. [Электрон. ресурс] URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7342211&newsearch=true&searchWithin="First%20Name":Mikhail&searchWithin="Middle%20Name":Yu.&sear-chWithin="Last%20Name":Orekhov (дата обращения: 15.02.2016).

8. *Орехов М. Ю.* Быстродействующая строковая система на C++ // Процессы управления и устойчивость. 2014. Т. 1, № 1. С. 363–368.

9. *Орехов М. Ю.* Применение подстроки в реализации быстродействующей строковой системы на C++ // Вестн. С.-Петерб. ун-та. Сер. 10. Прикладная математика. Информатика. Процессы управления. 2015. Вып. 2. С. 134–149.

10. International Standard ISO/IEC 14882:2014(E): Information Technology — Programming Languages — C++. 4th ed. 2014-12-15. § 12.8. Copying and moving class objects [class.copy] para. 31. [Электрон. ресурс] URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm&csnumber=64029 (дата обращения: 20.12.2015).

11. *Страуструп Б.* Язык программирования C++: спец. изд. / пер. с англ. С. Анисимова, М. Кононова; под ред. Ф. Андреева, А. Ушакова. М.: Бином-Пресс, 2004. 1104 с. (*Stroustrup B.* The C++ Programming Language)

12. *Бланшетт Ж., Саммерфилд М.* Qt4: программирование GUI на C++. 2-е изд., доп. / пер. с англ. С. Лунина, В. Казаченко. М.: Кудиц-Пресс, 2008. 736 с. (*Blanchette J., Summerfield M.* C++ GUI Programming with Qt 4)

References

1. Cormen T., Leiserson C., Rivest R., Stein C. *Introduction to Algorithms.* 3rd ed. Cambridge, Massachusetts, MIT Press, 2009, 1312 p. (Russ. ed.: Cormen T., Leiserson C., Rivest R., Stein C. *Algoritmy: Postroenie i Analiz.* Moscow, "Williams" Publ., 2013, 1328 p.)

2. Levitin A. *Introduction to the Design & Analysis of Algorithms.* Boston, Massachusetts, Addison-Wesley, 2003, 497 p. (Russ. ed.: Levitin A. *Algoritmy: Vvedenie v Razrabotku i Analiz.* Moscow, "Williams" Publ., 2006, 576 p.)

3. Pugh W. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 1990, vol. 33, no. 6, pp. 668–676.

4. Bezlepkin V. V., Kukhtevich V. O., Obraztsov E. P., Migrov Yu. A., Shaleninov A. A., Deulin A. A. *Programmno-tehnicheskij Kompleks "Virtual'nyj Energoblok AES s VVER" dlja Proverki Proektnyh Reshenij AES-2006* [*"Virtual Power Unit of Nuclear Power Plant with Pressurized Water Reactor" Instrume-ntal Complex Employment for Verifying Design Solutions*]. Available at: http://www.gidropress.podolsk.ru/files/proceedings/mntk2013/documents/mntk2013-168.pdf (accessed: 21.10.2015). (In Russian)

5. *Gorizonty Atoma (16.02.2013): Virtual'naja AES* [*Atom Perspectives* (*16.02.2013*): *Virtual NPP*] News Agency "Russian Atomic Community". Available at: http://www.atomic-energy.ru/video/40025 (accessed: 15.09.2015). (In Russian)

6. Kalinin D. V. *Mnogofunkcional'nyj Graficheskij Redaktor Videokadrov BPU v Sostave Modeli-*

rujush-hego Kompleksa "Virtual'nyj Energoblok" [*Multifunctional Graphics Editor of Soft Panels within "Virtual Power Unit" Computer Complex*]. Available at: http://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxzaGFyaW5nc3BhY2UxNnxneDo0YWI3NTRhM2JlZTMxODE5 (accessed: 20.02.2016). (In Russian)

7. Orekhov M. Yu. *Specialized Container and String Systems Employment in Vector Graphics Environment Development.* Available at: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7342211&new-search=true&searchWithin="First%20Name":Mikhail&searchWithin="Middle%20Name":Yu.&searchWithin="Last%20Name":Orekhov (accessed: 15.02.2016).

8. Orekhov M. Yu. Bystrodejstvujushhaja Strokovaja Sistema na C++ [C++ Quick-operating String System]. *Processy Upravlenija i Ustojchivost'* [*Control Processes and Stability*], 2014, vol. 1, no. 1, pp. 363–368. (In Russian)

9. Orekhov M. Yu. Primenenie Podstroki v Realizacii Bystrodejstvujuschej Strokovoj Sistemy na C++ [Substring Employment in C++ Quick-operating String System Implementation]. *Vestnik of Saint Petersburg University. Series 10. Applied mathematics. Computer science. Control processes*, 2015, issue 2, pp. 134–149. (In Russian)

10. *International Standard ISO/IEC 14882:2014(E): Information Technology — Programming Languages — C++.* 4th ed. 2014-12-15. § 12.8. Copying and moving class objects [class.copy] para. 31. Available at: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm&csnumber=64029 (accessed: 20.12.2015).

11. Stroustrup B. *The C++ Programming Language.* Special ed. Boston, Massachusetts, Addison-Wesley, 2000, 1040 p. (Russ. ed.: Stroustrup B. *Jazyk Programmirovanija C++.* Moscow, Binom Press, 2004, 1104 p.)

12. Blanchette J., Summerfield M. *C++ GUI Programming with Qt 4.* 2nd ed. Upper Saddle River, New Jersey, Prentice Hall, 2008, 752 p. (Russ. ed.: Blanchette J., Summerfield M. *Qt4: Programmirovanie GUI na C++.* Moscow, Kudits Press, 2008, 736 p.)

Статья рекомендована к печати проф. Л. А. Петросяном.

Статья поступила в редакцию 14 ноября 2015 г.

Статья принята к печати 25 февраля 2016 г.